4K

УДК 004.4'233+004.415.53

***А.В. Колчин*[1], *Р.В. Четвертак*[2]**
[1]Институт кибернетики имени В.М. Глушкова НАН Украины, г. Киев
Украина, 03680 МСП, пр. Академика Глушкова, 40, *kolchin_av@yahoo.com*
[2]ООО Айссофт, г. Киев, Украина
Roman.Chetvertak@iss.org.ua

Интерактивная система для анализа поведения формальных моделей программных систем

***A.V. Kolchin*[1], *R.V. Chetvertak*[2]**
[1]V.M. Glushkov Institute of Cybernetics of NAS of Ukraine, c. Kyiv
Ukraine, 03680 MSP, c. Kyiv, Glushkova ave., 40 , kolchin_av@yahoo.com
[2] ISS Soft Ltd., Kyiv, Ukraine
Roman.Chetvertak@iss.org.ua

Interactive System for Analysis of Formal Model Behavior

***О.В. Колчин*[1], *Р. В. Четвертак*[2]**

Інтерактивна система для аналізу поведінки формальних моделей програмних систем

[1]Інститут кібернетики імені В.М. Глушкова НАН України, м. Київ
Україна, 03680 МСП, пр. Академіка Глушкова, 40, *kolchin_av@yahoo.com*
[2]ООО Айссофт, м. Київ, Україна
Roman.Chetvertak@iss.org.ua

Цель работы – разработка инструментальных средств для автоматизации анализа и упрощения понимания поведения формальных моделей. Предложены методы локализации причин нарушения проверяемых свойств и вычленения релевантных элементов на основе аналитического установления фактических информационных связей, а также интерактивные методы декомпозиции сложных для понимания участков функционирования компонент и межкомпонентных взаимодействий. Разработана экспериментальная система, реализующая предложенные методы.
Ключевые слова: верификация, отладка.

The purpose of the work is development of methods and tools for automation of analysis and reduction of understanding complexity of the behavior of formal models. The methods for faults localization and relevant elements identification based on analysis of informational dependency are proposed. Also, interactive methods for decomposition of hard-to-understand behavior of intra- and inter-components interactions are described. Prototype of a software tool, which implements the methods, is developed.
Key words: model checking, debugging.

Мета роботи – розробка інструментальних засобів для автоматизації аналізу та спрощення розуміння поведінки формальних моделей. Запропоновані методи локалізації причин порушення властивостей та вичленення релевантних елементів на основі аналітичного встановлення фактичних інформаційних зв’язків, а також інтерактивні методи декомпозиції складних для розуміння ділянок функціонування компонент і міжкомпонентних взаємодій. Розроблена експериментальна система, що реалізує запропоновані методи.
Ключові слова: верифікація, відлагодження.

Введение

Возрастающая сложность логики поведения современных программных систем обостряет актуальность автоматизации проверки их правильности – верификации.

Существующие инструментальные средства верификации способны проверять на формальных моделях разрабатываемых систем выполнимость формул, выраженных темпоральной логикой. Это дает возможность проверки условий живости (например, полученный запрос всегда будет обслужен) и безопасности (например, лифт никогда не поедет с открытыми дверями) [1-4]. При обнаружении нарушения проверяемого свойства верификаторы, как правило, ограничиваются генерацией трассы-контрпримера, демонстрирующей достижимость ошибки, ее *симптомы* (например, [1-3, [5]). Такие трассы представляют собой линейную последовательность событий модели, что, к сожалению, дает слишком мало информации о *причинах* найденной ошибки и о том, что же нужно изменить в модели для ее устранения. Средства локализации причин ошибки и анализа последствий попыток ее исправления несправедливо получили недостаточное развитие и не удовлетворяют потребности промышленной разработки программного обеспечения. Основная задача данной работы – усовершенствовать методы и инструментарий отладки формальных моделей.

В разделе «формальная модель» приведены основные определения математического аппарата, раздел «вычисление информационных связей» описывает метод построения графа зависимостей формальной модели, раздел «анализ информационных связей» приводит описание методов навигации и визуализации графа зависимостей формальной модели, а также режима объяснения трасс. В завершении проиллюстрированы примеры, демонстрирующие достоинства метода и представлен краткий сравнительный анализ с аналогичными системами и методами.

Формальная модель

Ниже приведены основные формальные определения, необходимые для описания методов и их свойств.

Операционная семантика формальной модели может быть определена в терминах транзиционных систем. Пусть задано конечное множество атрибутов $A = v_1, v_2,..., v_n$ и пусть также для каждого атрибута $v_i \in A$ определена конечная область допустимых значений $D(v_i)$.

***Определение* 1.** Состоянием называется множество пар атрибутов и их значений вида $\{\bigcup_{i=0..|A|}(v_i = d_i) \mid v_i \in A,\ d_i \in D(v_i)\}$.

Назовем предусловием некоторую бескванторную формулу логики предикатов над атрибутами множества *A* и константами из соответствующих областей допустимых значений *D*. Назовем постусловием множество присваиваний вида *v* := *expr*(*s*), где $v \in A$, *s* – состояние, а $expr: s \rightarrow D(v)$ – некоторая функция над атрибутами состояния *s*.

***Определение* 2.** Переходом называется тройка вида (t, α, β), где *t* – имя перехода, α – его предусловие, а β – постусловие.

Семантика перехода такова: если в некотором состоянии *s* предусловие перехода *t* выполнимо, то модель может выполнить этот переход и перейти в новое состояние *s'=next(t,s)*, которое отличается от предыдущего значениями тех атрибутов, которым было выполнено присваивание новых значений в постусловии.

***Определение* 3**. Атрибутной транзиционной системой (АТС) называется шестерка вида $M = \langle S, s_0, T, A, D, F \rangle$, где *S* – конечное множество состояний, $s_0 \in S$ – начальное состояние, *T* – конечное множество переходов, *A* – атрибутов, *D* – соответствующие области допустимых значений, $F:(s,v) \rightarrow D(v)$ – функция, вычисляющая значение атрибута *v* в состоянии *s*.

Аналогично охраняемым командам Дейкстры [6] с помощью отношения *T* задаются возможные варианты дискретных детерминированных переходов из одного множества состояний в другое. АТС является удобным математическим аппаратом для описания поведения широкого спектра формальных моделей систем асинхронно-взаимодействующих процессов и их абстракций.

***Определение* 4.** Трассой в *M* из состояния s_i в состояние s_j называется такая последовательность состояний и переходов $s_i \xrightarrow{t_i} s_{i+1} \xrightarrow{t_{i+1}} s_{i+2} \ldots s_j$, что $s_k \in S \wedge t_k \in T$ для всех $k \in i..j$.

***Определение* 5.** Состояние *s* достижимо в модели *M*, если существует трасса, ведущая из начального состояния s_0 в *s*. Множество всех достижимых состояний будем обозначать *Reachable*(*M*, s_0).

Для выполнения перехода *t* из состояния *s* в новое состояние *s'* данная работа использует процедуру *transit*; ее подробное описание приведено в [7], а модификация выполнения предусловия – в [8]. Так, выполнение этой процедуры построит множество *R* атрибутов, значений которых (согласно лемме 1 [8]) достаточно для вычисления предусловия *t*, и (согласно лемме 4 [7]) если переход выполним, множество *W* атрибутов, которым осуществлялось присваивание, а также для каждого атрибута $w \in W$ множество *V[w]* атрибутов, которые входят в выражение, формирующее значение *w*: *(result,s',R,W,V) ← transit(s,t,s')*.

Примеры ее работы с выполнимыми переходами продемонстрированы на рис. 1, 2.

Рисунок 1	Рисунок 2
Предусловие перехода t1 : `(x=a+b ∧ y=0)` **Постусловие перехода t1 :** `a:=a-c; x:=0`	**Предусловие перехода t2 :** `(a>0 ∨ b=1)` **Постусловие перехода t2:** `a:=a-1; c:=b`
Вход: `transit( (a=2,b=0,c=1,x=2,y=0), t1, (a=2,b=0,c=1,x=2,y=0) )` **Выход:** `(result=`**`T`**`; s'=(a=1,b=0,c=1,x=0,y=0); R={x,a,b,y};W={a,x};V[a]={a,c},V[x]=∅)`	**Вход:** `transit( (a=2, b=0, c=1), t2, (a=2, b=0, c=1))` **Выход:** `(result=`**`T`**`; s'(a=1, b=0, c=0); R={a};W={a,c};V[a]={a},V[c]={b}`

Рисунок 1 – Пример 1 работы процедуры *transit*

Рисунок 2 – Пример 2 работы процедуры *transit*

На рис. 3, 4 продемонстрированы результаты работы процедуры *transit* с состояниями, из которых переходы невыполнимы.

Рисунок 3	Рисунок 4
Предусловие перехода t1 : `(x=a+b ∧ y=0)` **Постусловие перехода t1 :** `a:=a-c; x:=0)`	**Предусловие перехода t2 :** `(a>0 ∨ b=1)` **Постусловие перехода t2 :** `a:=a-1; c:=b`
Вход: `transit( (a=2,b=0,c=1,x=2,y=1), t1, (a=2,b=0,c=1,x=2,y=1) )` **Выход:** `(result=`**`F`**`; s'=(a=2,b=0,c=1,x=2,y=1); R={y}; W=∅; V[]=∅)`	**Вход:** `transit( (a=0, b=0, c=1), t2, (a=0, b=0, c=1))` **Выход:** `(result=`**`F`**`; s'=(a=0, b=0, c=1); R={a,b};W=∅, V[]=∅`

Рисунок 3 – Пример 3 работы процедуры *transit*

Рисунок 4 – Пример 4 работы процедуры *transit*

Необходимо отметить, что в результате работы процедуры *transit* множество *R*-атрибутов будет содержать необязательно все атрибуты, входящие в предусловие. Например, для вычисления $f_1 \wedge f_2 \wedge \ldots f_n$ при истинных $f_1, \ldots, f_{n-1}$ и ложном f_n, множество *R* будет содержать только атрибуты, входящие в f_n; аналогично только атрибуты, вхо-

дящие в f_n, будут в множестве *R* при вычислении $\sim(f_1 \vee f_2 \vee ...f_n)$ при ложных $f_1,...,f_{n-1}$ и истинном f_n. На рис. 5, 6 приведены примеры таких множеств, построенных для невыполнимых переходов.

Предусловие t3: `~(x=a+b) ∧ y=0`	
Вход – состояние S	**Выход – множество R**
a=0, b=0, x=0, y=0	(F, {x,a,b})
a=0, b=0, x=1, y=1	(F, {y})

Рисунок 5 – Примеры множества *R*

Предусловие t4: `x=0 ∨ ~(y=0 ∨ z=0)`	
Вход – состояние S	**Выход – множество R**
x=1,y=0,z=0	(F, {x,y})
x=1,y=1,z=0	(F, {x,z})

Рисунок 6 – Примеры множества *R*

Вычисление информационных связей

Данная работа не фокусируется на определении перестановочности переходов и фактических типах зависимости между ними (истинная, антизависимость, по выходным данным), однако результаты работы можно применить и для этих целей. Существующие работы, как правило, предполагают наличие зависимости между атрибутами *a* и *b* тогда, когда существуют два состояния, в которых различны только значения атрибута *a* и при этом в следующих состояниях значения атрибута *b* различны (например, как в определении ниже [9]).

***Определение* 6.** Атрибут *b* зависит от атрибута *a*, $(a,b) \in DR(M)$, тогда, и только тогда когда $\exists (s_1,s_2) \forall x(x \neq a \rightarrow s_1.x = s_2.x \wedge next(s_1).b \neq next(s_2).b)$.

Запись *‘s.x’* будет использоваться как сокращение от $F(s,x)$, т.е. обозначает значение атрибута *x* в состоянии *s*.

Для случаев, когда диапазон значений зависимых атрибутов включает более чем два элемента, такое определение может порождать существенную избыточность. Рассмотрим два примера (переходы модели в виде ‘имя перехода: предусловие => постусловие’):

```
t1: a=0 => b := 0
t2: a=1 => c := 0
t3: a=2 => d := 0
```

Рисунок 7 – Пример 1

```
T1: x=0 => b := 0
t2: y=0 => b := 1
t3: z=0 => b := 2
```

Рисунок 8 – Пример 2

t1: a_0=1 => b := 0
t2: a_1=1 => c := 0
t3: a_2=1 => d := 0

Рисунок 9 – Пример 1а

t1: x=0 => b_0 := 1
t2: y=0 => b_1 := 1
t3: z=0 => b_2 := 1

Рисунок 10 – Пример 2а

Первый пример (рис.7) показывает, что все атрибуты *{b, c, d}* зависят от *a*; второй (рис. 8) – что *b* зависит от атрибутов *{x,y,z}*. Однако эти же модели можно представить иным способом, заменив некоторые атрибуты множеством других, как показано на рис. 9 и 10 соответственно. Таким образом, в примере 1а атрибут *b* зависит от a_0, *c* от a_1, *d* от a_2; в примере 2а – b_0 от *x*, b_1 от *y*, b_2 от *z*. Сокращение числа зависимых элементов модели (вернее, более точное вычисление зависимостей) актуально не только для сокращения перебора при автоматической проверке свойств модели, но и для понимания зависимостей человеком.

Также такое определение зависимости порождает избыточную зависимость «самого от себя», когда, например, $s_1.a \neq s_2.a$ и $next(s_1).a \neq next(s_2).a$, но при этом атрибут *a* не встречается ни в предусловии, ни в постусловии перехода. Еще одним нюансом

определения зависимости является учет перехода, по которому вычисляется следующее состояние: достаточным будет существование перехода t, при котором либо выполнится условие $next(t,s_1).b \neq next(t,s_2).b$, либо t невыполним из s_2. Поэтому мы будем пользоваться уточненной спецификацией зависимости, определенной над парами вида *(атрибут=значение)*, которое включает условие различности пар и учитывает существование такого перехода.

***Определение* 7.** Пары $(a=v_a,b=v_b)\in DR(M)$ имеют информационную связь тогда и только тогда, когда

$$(a\neq b \vee v_a\neq v_b) \wedge \exists (s_1,s_2)\in Reachable(M, s_0):$$
$$\forall x(x\neq a \rightarrow s_1.x=s_2.x \wedge s_1.a=v_a \wedge \exists t(next(t,s_1).b=v_b \wedge$$
$$\wedge (next(t,s_2)=\varnothing \vee next(t,s_2).b\neq v_b))$$

***Определение* 8.** Графом зависимости называется ориентированный граф, вершинами которого являются пары вида *(атрибут=значение)*, а дуги отмечают факт истинной зависимости.

Метод построения графа зависимости. Для построения графа зависимости будут использоваться результаты R, W, $V[W]$ выполнения процедуры *transit(s,t,s')* следующим образом: если переход выполним, то для каждого $w\in W$ пара $(w, s'.w)$ будет соединена дугами, выходящими из всех пар $(r, s.r)$, $r\in R$, и из пар $(v, s.v)$, $v\in V[w]$. В случае, если переход невыполним, будет порождена специальная вершина с отметкой *(t=**F**)*, в которую будут входить дуги из всех пар $(r,s.r)$, $r\in R$. Необходимо отметить также, что дуги размечены именами переходов и различаются на три типа: смежные с вершинами из множеств R и V, и ведущие в специальные вершины вида *(t=**F**)*.

Теорема. Метод построения графа зависимости сохраняет информационные связи, порождаемые определением 7.

Доказательство: если значение некоторого атрибута не изменялось в постусловии перехода t, т.е. $b\notin W$, то $s.b=next(t,s).b$ для любого s согласно семантике перехода, следовательно, условие $s_1.b\neq s_2.b$ означает, что $b=a$, а это противоречит определению 7 (условию различности). Согласно лемме 1 [8], атрибутов множества R достаточно для интерпретации предусловия, и согласно лемме 4 [7] все атрибуты, которые входят в выражение, формирующее значение присваиваемых атрибутов, входят в соответствующие множества $V[]$. Таким образом, если некоторый атрибут в состоянии s не принадлежит ни R, ни $V[]$, то он не повлияет ни на какое значение из $next(t,s)$.

Таким образом, метод гарантирует, что если зависимость существует, то граф будет ее содержать. С другой стороны, граф может содержать избыточные связи, так как метод рассматривает только одно состояние, тогда как определение 7 подразумевает существование двух.

Анализ информационных связей

Далее предполагается, что информационная зависимость атрибутов модели представлена графом, построенным согласно определению 8. Граф может быть завершенным – в случае, если верификатору удалось построить все необходимые состояния модели, или незавершенным – в противном случае (если процесс был прерван или исчерпались ресурсы оперативной памяти). В последнем случае метод не претендует на полноту, тем не менее, может быть полезен как для частичного анализа, так и для интерактивного управления процессом верификации.

Интерактивная визуализация. Так как граф зависимостей может содержать довольно много элементов, необходимы удобные средства для его отображения и

навигации. Для этого разработана техника интерактивной визуализации. Так, изначально отображаются только вершины, выбранные пользователем по некоторым критериям (например, все вершины, содержащие выбранный атрибут, или вершины начального состояния модели и т.п.). Далее пользователь может управлять визуализацией с помощью команд influence и depend. Эти команды конструируются посредством графического интерфейса и имеют такую семантику:

– **influence(node)** добавляет отображение дуг, выходящих из вершины **node** и вершин, смежных по этим дугам. Используется для визуализации подграфа, который зависит непосредственно от данной вершины.

– **depend(node)** добавляет отображение дуг, входящих в вершину **node** и вершин, смежных по этим дугам. Используется для визуализации подграфа, который влияет на данную вершину (формирует ее).

Если у вершины не существует входящих дуг, это значит, что вершина принадлежит начальному состоянию; отсутствие выходящих дуг может означать наличие лишнего присваивания. Например, если этим присваиванием было выделение памяти, то, скорее всего, такая вершина – место утечки памяти.

Разметка. В разработку модели может быть вовлечена достаточно большая группа людей, выполняющих различные роли (исполнители, заказчики, архитекторы, технические эксперты и т.д.) в различных модулях и функциональностях, и как следствие, появляется потребность рассматривать модель с различных ракурсов и на разных уровнях абстракции. Например, на ранних стадиях разработки чаще всего анализируется поток управления, в то время как при анализе причин дефекта более интересна история записи и чтения некоторых данных. Часто причиной дефектов становится неправильное взаимодействие функциональностей (feature interaction). Анализ таких дефектов весьма затруднен: реализация одной функциональности может быть распределена по различным компонентам; трудоемкость идентификации релевантных к ней элементов значительна.

Предлагаемая техника разметки призвана облегчить анализ модели и предоставить ее компактную визуализацию на различных уровнях абстракции. Так, каждый элемент модели может быть размечен множеством меток, обозначающих признак логической принадлежности к некоторой группе. Такой группой может стать модуль, функциональность, процедура, цикл, и т.п. Визуализация графа позволяет скрывать/отображать все элементы выбранной группы. Разметка осуществляется командой mark, а управление командами show, hide, collapse, expand и interact, которые имеют такую семантику:

– **mark(node** или **tag, new_tag)** – устанавливает отметку **new_tag** на вершину **node** или на все вершины, атрибуты которых размечены отметкой **tag**. В качестве отметки могут также быть использованы имена атрибутов или модулей. Используется для группирования вершин с последующим обращением к ним по имени отметки.

– **show(tag)** добавляет отображение всех вершин, размеченных отметкой **tag**, и дуг между ними.

– **hide(tag)** скрывает все вершины, размеченные отметкой **tag**, и дуги между ними.

– **collapse(tag)** свертывает все вершины, размеченные отметкой **tag**, и дуги между ними в одну вершину, также в эту вершину перенаправляются все внешние дуги. При дублировании дуги объединяются в одну.

– **expand(tag)** раскрывает все вершины, размеченные отметкой **tag**, и дуги между ними; внешние дуги расставляются на свои места.

– **interact(tag_1, tag_2)** раскрывает все дуги, соединяющие вершины из множества **tag_1** с вершинами из множества **tag_2**, а также добавляет отображение смежных с этими

дугами вершин. Используется для визуализации межкомпонентного взаимодействия, а также участков взаимодействия функциональностей.

Режим объяснения трассы. Команда **explain_trace** отображает трассу и ее связи с графом поведения модели. Каждый переход на трассе разделен на две части: пред- и постусловие. Вершины основного графа информационных связей соединены дугами с пред- и постусловиями переходов по принципу: если атрибуту *a* присваивалось значение *v*, то из постусловия выходит дуга в вершину *(a,v)*; аналогично отображаются дуги, входящие в предусловия. Также доступен интерактивный режим объяснения трассы, в котором трасса визуализируется не полностью, а по частям. Так, изначально отображаются два элемента трассы – первый и последний переходы. Далее выполняются команды пользователя. При этом визуализация внутренних элементов трассы сопровождается «отодвиганием» элементов, расположенных ниже. Пользователь такой техники объяснения трассы имеет возможность нажатием одной кнопки получить ответы на самые актуальные при поиске причин дефекта вопросы – когда (в каком месте на трассе) атрибут получил текущее значение и где оно используется еще (в каком месте на основном графе). Мотивацией такого подхода был дефект, обнаруженный верификатором на трассе длиной около 1000 переходов. Как оказалось в последствии (и это было проверено экспериментально с помощью описанного метода), причина дефекта объяснялась достаточно легко – была упущена инициализация нужного атрибута, тогда как его использование предусмотрено процедурой, вызов которой лежал на упомянутой глубине трассы.

Примеры

Подграфы выполнения перехода модели. Ниже проиллюстрированы подграфы, которые построены для выполнения переходов из примеров 1 и 2 (см. рис. 1 и 2): рис.11 и 13 – для режима объяснения трассы, рис.12 и 14 – в общем графе зависимостей соответственно.

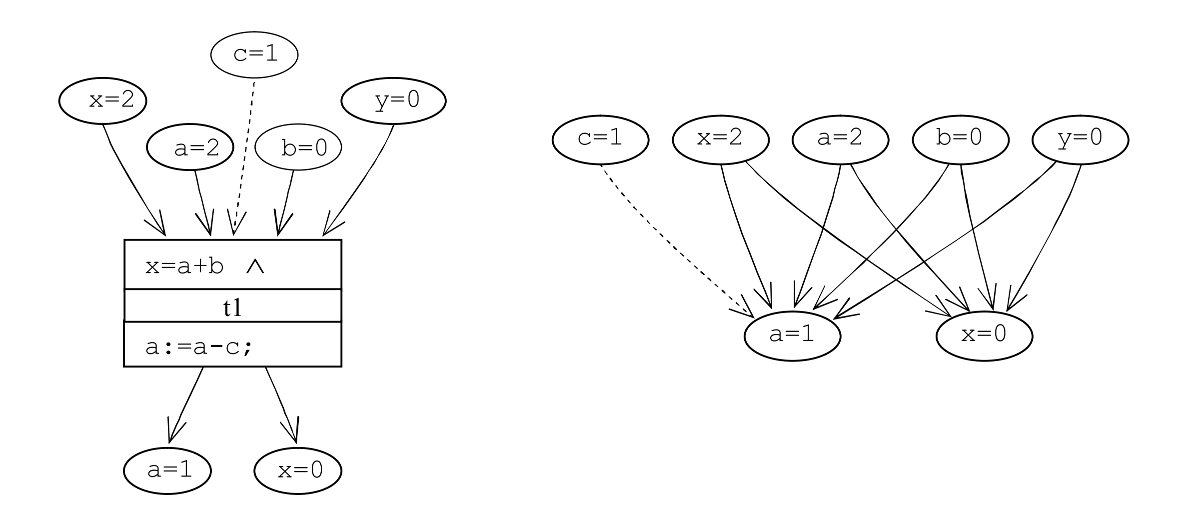


Рисунок 11 – Подграф примера 1 в режиме объяснения трассы

Рисунок 12 – Подграф примера 1 в общем графе зависимостей

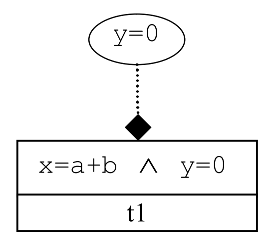


Рисунок 13 – Подграф примера 2 в режиме объяснения трассы

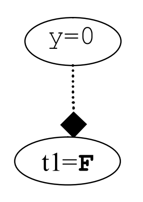


Рисунок 14 – Подграф примера 2 в общем графе зависимостей

Локализация причины дефекта и его устранение в режиме объяснения трассы. Следующий пример демонстрирует анализ причины найденного тупика (deadlock) и его устранение. Трасса, ведущая из начального состояния к состоянию тупика, представлена на рис. 15. На рис. 16 показан подграф в режиме объяснения трассы.

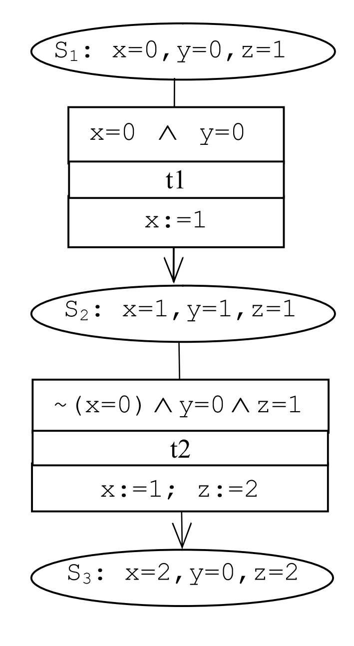


Рисунок 15 – Трасса из начального состояния в состояние тупика

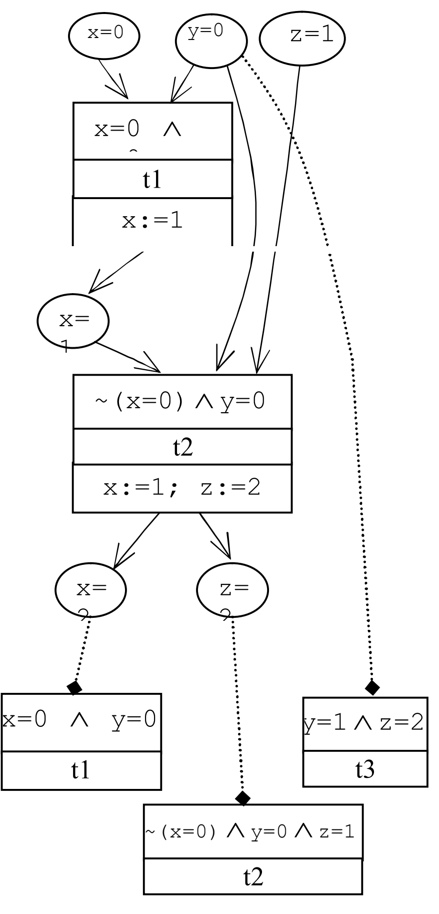


Рисунок 16 – Подграф в режиме объяснения трассы

Из него видно, что причиной невыполнения перехода *t1* является вершина *x=2*, *t2* – *z=2* и *t3* – *y=0* соответственно. Пусть достоверно известно, что продолжением трассы должен быть переход *t3*, и его предусловие верно. Так как в переход *t3* дуга ведет (рис. 17) из вершины начального состояния, (*y=0* не имеет входов), то это означает, что где-то на трассе упущено присваивание значения 1 атрибуту *y*. Необходимо отметить, что если такое присваивание вставить в постусловие перехода *t1*, то следующий за ним переход *t2* окажется невыполнимым. Факт использования переходом *t2* вершины *y=0* виден на подграфе ее влияния (рис. 18).

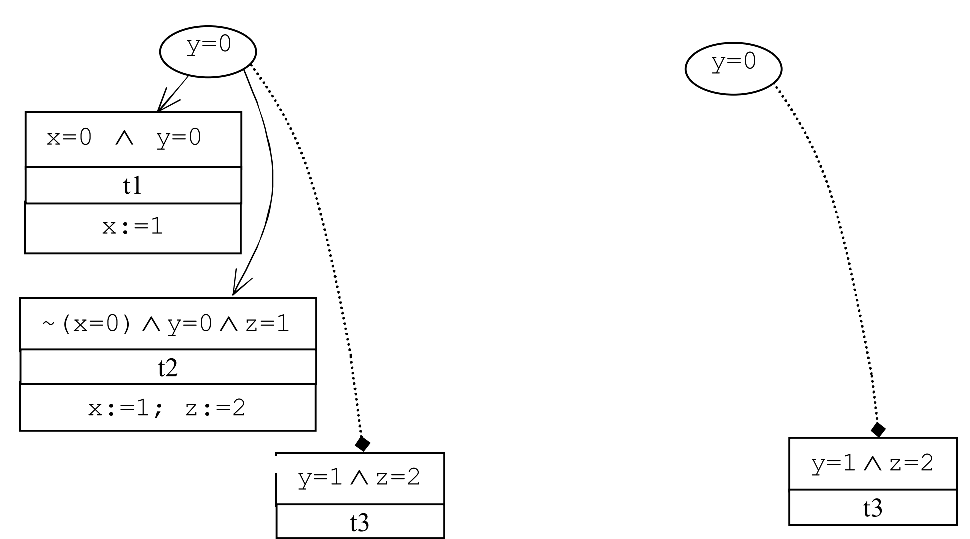


Рисунок 18 – Подграф влияния вершины *y=0*

Рисунок 17 – Подграф объяснения причины невыполнимости перехода *t3*

Таким образом, необходимое присваивание можно безболезненно добавить в постусловие перехода *t2* (рис. 19).

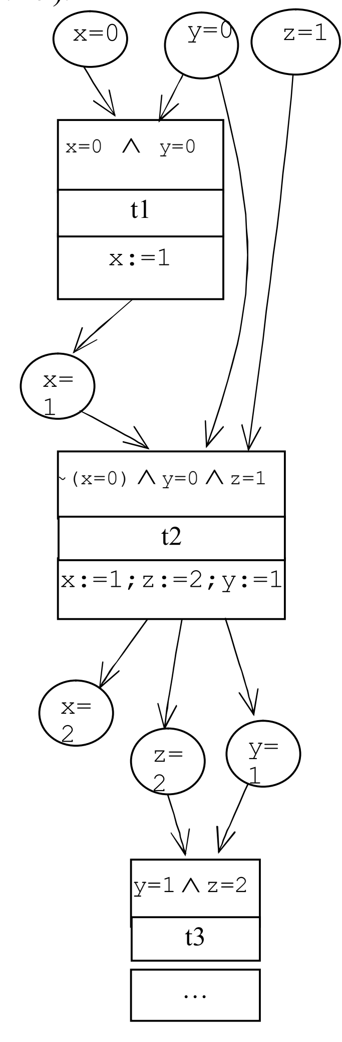


Рисунок 19 – Подграф в режиме объяснения трассы: тупик устранен

Сравнительный анализ с существующими системами

В качестве примеров, развивающих методы анализа причин дефектов в трассе-контрпримере, которая выдана верификатором, можно привести работы [10-13]. По сути, в их основе лежат алгоритмы анализа различий между трассами, которые содержат ошибку, и трассами, не содержащими ее. Представительные методы анализа графа информационных зависимостей модели описаны в [9], [14], [15]. Метод [9] предполагает дополнительным входом описание ожидаемых информационных связей и осуществляет статическую проверку со связями, выявленными в переходах модели. Аналогично метод [14] выполняет структурную проверку на формальной модели спецификаций функциональных зависимостей. Работа [15] применяет метод расслаивания (slicing) и выполняет проверку связей только на тех слоях, которые могут иметь отношение к проверяемым свойствам. Похожим методом визуализации графа зависимостей можно назвать [16]. Его основная идея заключается в группировании вершин графа по структурным признакам принадлежности процедурам и циклам. Также необходимо отметить схожую с данной работой разработку для отладки программ [17], ставшую призером «Eclipse Hot New Products Showcase» в 2011 году [18].

Основным отличием от существующих систем и методов можно назвать технику локализации причины невыполнения перехода [7, 8], а также возможность определения факта использования значения атрибута. Эта особенность значительно облегчает навигацию по графу зависимостей модели и поиск причины дефектов.

Выводы

Усовершенствовано определение зависимости атрибутов формальной модели. Уточнения позволяют сократить перебор при проверке свойств и более детально анализировать причинно-следственные связи значений атрибутов модели.

Предложенная техника интерактивной навигации по графу зависимостей позволяет избежать одновременного отображения всех элементов графа, строить его визуализацию компактно, а также оперативно управлять процессом верификации, что особенно важно для больших систем. Предложенная техника разметки дает возможность удобно декомпозировать модель на составляющие компоненты, рассматривать ее поведение на разных уровнях абстракции, строить ее проекции на интересующие функциональности и взаимодействия между ними. Это позволяет разным исполнителям-экспертам по различным функциональностям оценивать адекватность реализации задуманному, а также облегчить встраивание новых функциональностей в существующую модель.

Техника анализа трасс упрощает отладку моделей и установление причин дефектов, а также их результативное устранение.

Разработана экспериментальная вычислительная система, реализующая методы построения графа зависимостей и его визуализацию в среде Eclipse.

Планируется усовершенствовать методы разметки путем формирования множества вершин по логическим (например, по принципу удовлетворения некоторому свойству) и структурным (выполнению процедуры и т.п.) свойствам. Также интерес вызывает применение метода установления информационных связей к генерации тестов, в частности минимизации тестовых наборов путем устранения перебора независимых участков трасс.

Литература

1. Обзор современных систем и методов верификации формальных моделей / А.В. Колчин, А.А. Летичевский, С.В. Потиенко [и др.] // Проблемы программирования. – 2012. – № 4. – 13 с.
2. Software model checking / R. Jhala, R. Majumdar // ACM Comput. Surv. – 2009. – Vol. 41 (4). – 54 p.
3. Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем / Карпов Ю.Г. – Санкт-Петерберг : БХВ-Петербург, 2010. – 552 с.
4. Emerson E. Temporal and modal logic / E. Emerson // J. van Leeuwen editor: Handbook of Theoretical Computer Science, Elsevier. – 1990. – P. 997-1072.
5. Holzmann G. The SPIN Model Checker: Primer and Reference Manual / Holzmann G. – Addison-Wesley Professional, 2003. – 596 P.
6. Dijkstra E. Guarded commands, nondeterminacy and formal derivation of programs / E. Dijkstra // Communications of the ACM. – 1975. – Vol. 18, № 8. – P. 453-457.
7. Колчин А.В. Автоматический метод динамического построения абстракций состояний формальной модели / А.В. Колчин // Кибернетика и системный анализ. – 2010. – № 4. – С. 70-90.
8. Колчин А.В. Оптимизация проверки выполнимости переходов при верификации формальных моделей / А.В. Колчин // Проблемы программирования. – 2012. – № 2-3. – 10 с.
9. Jackson D. Aspect: detecting bugs with abstract dependences / D. Jackson // ACM transactions on software engineering and methodology. – 1995. – Vol. 4, № 2. – P. 109-145.
10. From Symptom to Cause: Localizing Errors in Counterexample Traces / T. Ball, M. Naik, S. Rajamani // In Proc. of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 2003. – Vol. 38. – P. 97-105.
11. Ilan Beer. Explaining Counterexamples Using Causality / Ilan Beer, Shoham Ben-David and oth. // Formal Methods in System Design. – 2012. – № 40. – P. 20-40.
12. Groce A. What Went Wrong: Explaining Counterexamples / A. Groce, W. Visser // In SPIN'03 Proc. of the 10th Int. Conf. on Model checking software. – 2003. – P. 121-136.
13. Masri W. Fault localization based on information flow coverage / W. Masri // Software Testing, Verification and Reliability. – 2010. – № 20. – P. 121-147.
14. Peischl B. An Abstract Operational Framework for Dependence Models in Software Debugging / B. Peischl, S. Soomro, F. Wotawa // In Proc. of the 2011 IEEE Fourth Int. Conf. on Software Testing, Verification and Validation Workshops. – 2011. – P. 597-606.
15. Cortesi A. Dependence Condition Graph for Semantics-based Abstract Program Slicing / A. Cortesi, R. Halder // In Proc. of the 10th Workshop on Language Descriptions, Tools and Applications. – 2010. – № 4. – 9 p.
16. Balmas F. Displaying dependence graphs: a hierarchical approach / F. Balmas // J. of software maintenance. – 2004. – № 16. – P. 151-181.
17. [Элекронный ресурс]. – Режим доступа : http://www.chrononsystems.com/products/chronon-time-travelling-debugger
18. [Электронный ресурс]. – Режим доступа : http://www.eclipse.org/org/press-release/20110324_showcasewinners.php

Literatura

1. Kolchin A.V. Problemy programmirovanija. 2012. № 4. S. 13.
2. Jhala R. ACM Comput. Surv. Vol. 41 (4). 2009. 54 p.
3. Karpov Ju.G. Model Checking. Verifikacija parallel'nyh i raspredelennyh programmnyh sistem. BHV-Peterburg. 2010. 552 S.
4. Emerson E. Handbook of Theoretical Computer Science. Elsevier. 1990. P. 997-1072.
5. Holzmann G. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional. 2003. 596 p.
6. Dijkstra E. Communications of the ACM. 1975. Vol.18. № 8. P. 453-457.
7. Kolchin A.V. Kibernetika i sistemnyj analiz. 2010. № 4. S. 70-90.
8. Kolchin A.V. Problemy programmirovanija. 2012. №2-3. 10 s.
9. Jackson D. ACM transactions on software engineering and methodology. 1995. Vol.4. № 2. P. 109-145.
10. Ball T. In Proc. of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2003. Vol. 38. P. 97-105.
11. Ilan Beer. Formal Methods in System Design. 2012. № 40. P. 20-40.
12. Groce A. In SPIN'03 Proc. of the 10th Int. Conf. on Model checking software. 2003. P. 121-136.

13. Masri W. Software Testing, Verification and Reliability. 2010. № 20. P. 121-147.
14. Peischl B. In Proc. of the 2011 IEEE Fourth Int. Conf. on Software Testing, Verification and Validation Workshops. 2011. P. 597-606.
15. Cortesi A. In Proc. of the 10th Workshop on Language Descriptions, Tools and Applications. 2010. №4. 9 p.
16. Balmas F. J. of software maintenance. 2004. № 16. P. 151-181.
17. http://www.chrononsystems.com/products/chronon-time-travelling-debugger
18. http://www.eclipse.org/org/press-release/20110324_showcasewinners.php

A.V. Kolchin, R.V. Chetvertak

Interactive System for Analysis of Formal Model Behavior

This paper describes new methods for formal models behavior debugging and analysis based on construction of data dependency graph. Original methods of minimization of attributes subset, which is needed for transitions applicability computation, are used for the dependency construction. The dependency relation is improved by defining it in a more precise way – we use dependency between pairs like (attribute=value). This technique allows to analyze cause-effect relation in more detail and also to reduce search space.

Proposed technique of interactive navigation via dependency graph allows to avoid simultaneous visualization of whole graph elements and to build the visualization more compact, and also to control the verification process, which is important for large systems.

Described technique of the graph elements marking allows to make decomposition of a model, to view its behavior on different levels of abstraction, to build projections on functionality-of-interest and to view feature interactions.

Also, the method for trace analysis, which is helpful for formal models debugging and identification of defects reasons, is developed. The method also essentially simplifies defects fixing.

A special experimental software tool is developed, which utilizes the proposed methods. Graphical visualization is implemented in Eclipse environment.

Статья поступила в редакцию 01.06.2012